CSE 341, Winter 2008, Assignment 4
Due: Wednesday 20 February, 8:00AM

Last updated: February 7

You will write 8 Scheme functions (not counting helper functions) and 1 Scheme macro.
Begin by downloading `hw4.scm` from the course website. Add to this file to complete your homework.

Provided Code:
The code at the top of the file uses DrScheme's graphics library to provide a simple and entertaining (?) outlet for your streams. You need not understand this code (though it is not too complicated). This is how you use it:

- `(open-window)` returns a graphics window you can pass as the first argument to `place-repeatedly`.
- `(place-repeatedly window pause stream n)` uses the first `n` values produced by `stream`. Each stream element must be a pair where the first value is an integer between 0 and 5 inclusive and the second value is a string that is the name of an image file (e.g., `.jpg`). (7 image files that will work well are available on the course website). Every `pause` seconds (where `pause` is a decimal, i.e., floating-point, number), the next stream value is retrieved, the corresponding image file is opened, and it is placed in the window using the number in the pair to choose its position in a 2x3 grid as follows:

0	1	2
3	4	5

- If you are using DrScheme on a remote computer over the network, the time necessary to display images might increase considerably.

The code at the bottom of the file provides an example use of `place-repeatedly`. This code requires you complete several of the problems, of course. You should be able to figure out how this testing code should behave. Of course, this test case may not be sufficient.

Warning: Only the first three problems are "warm-up" exercises for Scheme. Subsequent problems dive into streams, memo-tables, and macros, which are nontrivial concepts.

Problems:

1. Write a function `sequence` that takes 3 arguments `low`, `high`, and `stride`, all assumed to be numbers. `sequence` produces a list of numbers from `low` to `high` (including `low` but not `high`) separated by `stride` and in sorted order. Sample solution: 4 lines. Examples:

Call	Result
`(sequence 3 11 2)`	`(3 5 7 9 11)`
`(sequence 3 8 3)`	`(3 6)`
`(sequence 3 2 1)`	`()`

2. Write a function `string-append-map` that takes a list of strings `lst` and a string `suffix` and returns a list of strings. The i^{th} element of the output should be the concatenation of i^{th} element of `lst` and `suffix`. Use library functions `map` and `string-append` (see the language specification as necessary). Sample solution: 2 lines.

3. Write a function `list-nth-mod` that takes a list `lst` and a number `n`. If `n` is negative or `lst` is empty, use `error` to print an appropriate error message. (Write `(error "blah")` to terminate computation with message `"blah"`.) Otherwise, return the i^{th} element of the list where we *count from zero* and i is the remainder produced when dividing `n` by the length of the list. Library functions `length`, `remainder`, `car`, and `list-tail` are all useful. Sample solution is about 7 lines; shorter solutions are possible.

4. Write a stream `dan-then-ben`, where the elements of the stream alternate between `"dan.jpg"` and `"ben.jpg"` (starting with `"dan.jpg"`). More specifically, `(dan-then-ben)` should produce a pair of `"dan.jpg"` and a thunk that when called produces a pair of `"ben.jpg"` and a thunk that when called... etc. Hint: Use 2 mutually recursive functions. Sample solution: 4 lines.

5. Write a function `stream-add-zero` that takes a stream `s` and returns another stream. If `s` would produce v for its i^{th} element, then `(stream-add-zero s)` would produce the pair `(0 .` v`)` for its i^{th} element. Sample solution: 4 lines. Hint: Use a thunk that when called uses `s` and recursion. Note: You can test `(stream-add-zero dan-then-ben)` with `place-repeatedly`.

6. Write a function `stream-for-n-steps` that takes a stream `s` and a number `n`. It returns a list holding the first `n` values produced by `s` in order. Sample solution: 5 lines. Note: You can test your streams with this function instead of the graphics code.

7. Write a function `cycle-lists` that takes two lists `numbers` and `filenames` and returns a stream. The first is a list of numbers and the second is a list of strings; the lists may or may not be the same length. The elements produced by the stream are pairs where the first part is from `numbers` and the second part is from `filenames`. The stream cycles forever through the lists. For example, if `numbers` is `(1 2 3)` and `filenames` is `("a" "b")`, then the stream would produce, `(1 . "a")`, `(2 . "b")`, `(3 . "a")`, `(1 . "b")`, `(2 . "a")`, `(3 . "b")`, `(1 . "a")`, `(2 . "b")`, etc.

   Sample solution is 6 lines and may be significantly more complicated than the previous stream problems. Hints: Use one of the functions you wrote earlier. Use a recursive helper function that takes a number `n` and calls itself with `(+ n 1)` (inside a thunk).

8. Write a function `mtf-assoc` (short for "move-to-front assocation") that takes one argument `lst`, which should be a list of pairs where the first parts of the pairs are numbers. `mtf-assoc` should return a function that takes a number `n` and returns `#f` if no pair in `lst` has `n` for its first component else it returns the first pair in `lst` with the number `n`.

   However, `mtf-assoc` must use a memoization-like technique to "optimize" for repeated queries as follows: `mtf-assoc` will return a closure that has a binding for the list in it. Whenever, this closure returns a pair it also *mutates* (with `set!`) its binding for the list so that the returned pair is *moved* to the front of the list. For example, if the initial list is `((1 . "a") (2 . "b") (3 . "c"))` and 2 is passed to the closure, the code should make the new list `((2 . "b") (1 . "a") (3 . "c"))` and then use `set!` so that the next time the closure is used it uses this new list. Sample solution: 10 lines.

   Hints: Use a local recursive helper function that keeps track of what list elements it has already seen (in reverse order) and then use `append`, `cons`, and `reverse` (all provided by the language) to build the new list. For testing, you might put in a print statement in your helper function so you can tell how many times it iterates.

   Note: It would be more efficient to use `set-cdr!` (in addition to `set!`) rather creating a new list each time. You may do this if you wish. (Though if you do, you should *copy* the initial list passed to `mtf-assoc`, else your mutation messes up callers who expect `lst` to stay unchanged.)

9. Write a macro `orelse` such that `(orelse e1 e2)` returns the result of `e1` unless that result is `#f` in which case it returns the result of `e2`. Evaluating `(orelse e1 e2)` must evaluate `e1` exactly once. If the result is `#f` it must evaluate `e2` exactly once, else it must not evaluate `e2`.

10. **Challenge Problem:** Write `cycle-lists-challenge`. It should be equivalent to `cycle-lists`, but its implementation must be more efficient. In particular, for each time the stream produces a new value, the code must perform only two `car` operations and two `cdr` operations, including operations performed by any function calls. So, for example, you cannot use `length` because it uses `cdr` multiple times to compute a list's length.

11. **Challenge Problem:** Write `orelse-challenge`, which is like `orelse` except it should take 2 *or more* arguments. It must evaluate no argument more than once and not evaluate any arguments after the first one that evaluates to something other than `#f`.

Assessment: Your solutions should be correct, in good style (including indentation and line breaks), and using features we have used in class. In particular, *only problem 8 should use mutation.*

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw4.scm`, where `lastname` is replaced with your last name.
- The first line of your `.scm` file should be a Scheme comment with your name and the phrase `homework 4`.
- Go to `https://catalysttools.washington.edu/collectit/dropbox/djg7/1359` (link available from the course website), follow the "Homework 4" link, and upload your file.